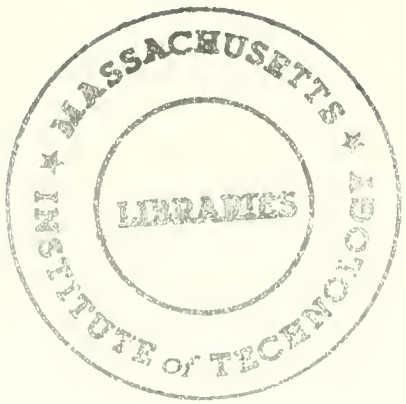
MASSACHUSETTS
INSTITUTE of TECHNOLOGY
LIBRARIES

WORKING PAPER
ALFRED P. SLOAN SCHOOL OF MANAGEMENT

HIERARCHICAL TIMESTAMPING ALGORITHM

Meichun Hsu
Stuart E. Madnick

April 1987 1876-87

MASSACHUSETTS
INSTITUTE OF TECHNOLOGY
50 MEMORIAL DRIVE
CAMBRIDGE, MASSACHUSETTS 02139

HIERARCHICAL TIMESTAMPING ALGORITHM

Meichun Hsu
Stuart E. Madnick

April 1987 1876-87

Hierarchical Timestamping Algorithm

By

Meichun Hsu
Harvard University
Cambridge MA 02138

and

Stuart E. Madnick
Massachusetts Institute of Technology
Cambridge, MA 02139

Acknowledgement: Work reported herein has been supported, in part, by the Naval Electronic Systems Command through contracts N0039-83-c-0463 and N0039-85-c-0571.

ABSTRACT

The Hierarchical Timestamping Algorithm is proposed for handling database concurrency control. By analyzing transaction conflicts and partitioning the database into hierarchical partitions to which transactions will access discriminantly using different synchronization protocols, the algorithm can offer significant performance gain. It also reduces the need for transactions to leave traces (e.g., locks, timestamps) when accessing a data element.

It is shown that the algorithm is correct in terms of transaction serializability. This is done by showing that the algorithm enforces a topological order among transactions. This research advocates the potential benefit of application analysis in enhancing performance of concurrency control algorithms where the level of concurrency is vital to system performance.

1. Introduction

The two basic approaches to database concurrency control are the two-phase locking technique [Eswaran76] and the timestamp ordering technique [Bernstein80, Reed78]. Both techniques endorse serializability as their criterion of correctness. While having the advantages of being deadlock free and in no need of unlock instructions when a transaction is finished, the timestamping algorithm has the drawback of rigidly obeying the order of timestamps which have been assigned to transactions without much consideration of the static or the dynamic nature of actual interferences among transactions. In this paper, the Hierarchical Timestamping (HTS) approach is described which makes unique use of the nature of the application conflicts. By decomposing the database into partitions to which the transactions will access discriminantly, the approach can offer significant performance gain in handling database concurrency control. Equally important is its prospect in reducing the need for a transaction to leave a "trace" (e.g., a lock or a timestamp) when accessing data elements.

Conflict analysis among transactions has been proposed in the research of SDD-1 [Bernstein80, Bernstein81] as a vehicle to discover, a priori, certain (static) conflict patterns among transaction classes that may enable a more flexible timestamp protocol (e.g., Protocol 1 in SDD-1's terminology) to be used. However, with an orientation towards a distributed database system where timestamps are not stapled with data elements, the SDD-1 approach stops short of developing a more generalized timestamping algorithm that takes better advantage of information that may be provided via conflict analysis. Along a different dimension, a novel method has been presented in [Viemont82] which blends timestamp ordering and two-phase locking in one and chooses to switch to one or the other at the most opportune time so as to increase level of concurrency. The multi-version timestamping algorithm has also been developed and shown to provide a higher level of concurrency than the conventional single-version timestamping algorithm [Reed78, Reed79, Bernstein83]. Theoretical aspects of multi-version databases are discussed in [Bernstein83, Papadimitriou84], while one-previous-version concurrency control methods in

[Bayer80, Garcia-Molina82] and multiple-previous-version methods in [Stearns81, Chan82, Chan85]. Simulation studies of multi-version methods have been reported in [Harder86, Carey86] and the results are generally favorable.

Using conflict analysis and multi-version database as vehicles, the algorithm proposed in this paper, called the Hierarchical Timestamping Algorithm, or the HTS algorithm, allows a transaction to use an "array" of timestamps, rather than a single timestamp, to synchronize its accesses. Which timestamp a transaction will use to synchronize an access to a particular data element depends on which "data partition" the data element belongs to. The method is based on the hierarchical database decomposition proposed in [Hsu83]. In comparison, the structural locking protocol [Silberschatz80, Kedem83] is a non-two-phase locking protocol which aims at increasing level of concurrency by reducing the amount of time the locks on the high-level nodes of a tree must be held by each transaction. It requires that the transactions access the nodes of the tree in certain sequence and it involves lock and unlock protocols for each access to the high-level nodes of the tree. The hierarchy referred to in their study is entirely different from the kind of hierarchy discussed in this paper.

Before going into the detailed mechanism of the algorithm, the basic idea behind the algorithm can be illustrated by referring to an example schedule of read and write steps from three transactions t_0,t_1, and t_2:

$t_0(W,b)$, $t_1(R,a)$, $t_2(R,b)$, $t_1(W,b)$, $t_2(W,c)$

where $t_1(R,a)$ refers to a request from transaction t_1 to read data element a, and $t_1(W,b)$ to write data element b, and so on. Suppose the timestamp of t_1 is smaller than that of t_2. Using the basic timestamping algorithm, t_1 will be denied permission to write data element b (since at that time b is stamped with a t_2 read timestamp which is greater than that of t_1's), and forced to abort. Under our proposed method, however, if the data partition containing data element a is related to that containing data element b in a certain way (the exact nature to be explained later), then the algorithm allows transaction t_2 to access b with a 'pseudo' read timestamp TS

earlier than t_1's timestamp so that t_1's write request can proceed without being delayed or aborted. Furthermore, this pseudo read timestamp TS may be engineered to be early enough so that no transactions will any longer write b with even earlier timestamps, eliminating the need to leave TS with b as a read timestamp. Therefore this read access can be accomplished without having to leave any "trace" for concurrency control purposes. With these properties, the proposed algorithm has tbe potential of increasing the level of concurrency while reducing the costly overhead of leaving access traces.

The organization of this paper is as follows. In the following section, the Hierarchical Timestamping Algorithm is described in detail. The proof of correctness is given in section three. In section four, we discuss the optimality aspect of the algorithm. Section five concludes the paper.

2. The Hierarchical Timestamping Algorithm

The HTS Algorithm requires the decomposition of a database into a number of data partitions. We construct a data partition hierarchy which is basically a partial order of the data partitions subject to certain constraints. The Hierarchical Timestamping Algorithm is a timestamp-based concurrency control algorithm parameterized by a chosen data partition hierarchy and its corresponding transaction classification. When the database decomposition consists of a single data partition, the HTS Algorithm degenerates to the conventional multi-version timestamping algorithm.

We will in this section first discuss the method of transaction analysis and the mechanism for constructing a database partition hierarchy. Then in Section 2.2 we present the algorithm itself.

2.1. Transaction Analysis

Let a database be decomposed into a number of *data partitions*. The purpose of transaction analysis in our algorithm is to facilitate *correct* construction of a *data partition hierarchy* on which the hierarchical timestamp protocol is based. The data partition hierarchy, constructed

off-line, is basically a partial order of the data partitions subject to additional graph-theoretic constraints and constraints related to transaction analysis. Note that given any data base partition and any set of transactions, a correct data partition hierarchy can be found and the hierarchical timestamp protocol applicable. However, performance and optimality of the protocol (e.g., whether it can outperform the basic multi-version timestamp algorithm) depends on an intelligent choice of database deomposition and data partition hierarchy. In this section, the mechanics of constructing a correct data partition hierarchy is discussed.

2.1.1. Transaction analysis against a database partition

Analysis of a set of update transactions T_s against a database consisting of a set of data partitions results in a *data partition graph*, where the nodes are the data partitions, and the arcs are assigned in such a way that there is an arc from a data partition D_i to another data partition D_j if and only if one can find a potential transaction in the database system that updates data elements in D_i and accesses (i.e., reads or writes) data elements in D_j. That is, $D_i \rightarrow D_j$, $i \neq j$, indicates that there exist transactions in the system that would potentially link updates of data elements in D_i to the content of data elements in D_j.

Definition. Let T_s be a set of *update* transactions to be performed on a database D. Let P be a decomposition of D into *data partitions* $D_1, D_2, \ldots, D_n$. A *data partition graph* of P w.r.t. T_s is a digraph denoted as $DPG(P, T_s)$ with nodes corresponding to the data partitions of P and a set of directed arcs joining these nodes such that, for $i \neq j$, $D_i \rightarrow D_j$ iff there exists a transaction $t \in T_s$ s.t. $w(t) \bigcap D_i \neq \phi$ and $a(t) \bigcap D_j \neq \phi$, where $w(t)$, $r(t)$ and $a(t)$ are the write set, the read set and the access set of transaction t. (The access set $a(t)$ is the union of $r(t)$ and $w(t)$.)

The data partition graph is a tool for capturing the pattern of conflict among transactions to be run in the system. Note that, in our algorithm, *there is no need for read-only transactions to participate in the transaction analysis*, eliminating the difficulties of pinning down, a priori, the nature of all ad hoc queries.

2.1.2. Constructing a data partition hierarchy

Once a data partition graph is defined for a database, a data partition hierarchy, which assigns a partial order to the set of data partitions, can be derived as follows.

Definition. Given a data decomposition P and a data partition graph $DPG(P,T_s)$, a *data partition hierarchy*, denoted as $DPH(P,T_s)$, is any acyclic graph where nodes are data partitions in P, and arcs between partitions, denoted as $\rightarrow$, such that

(1) $DPH(P,T_s)$ is a semi-tree (a semi-tree is an acyclic digraph where there exists at most one path between any pair of nodes,) and

(2) If there exists a directed path between D_i and D_j in $DPG(P,T_s)$, then there exists a directed path between D_i and $D_j$in $DPH(P,T_s)$.

Note that a data partition hierarchy is necessarily a transitive reduction: there are no transitive arcs in $DPH(P,Tu)$. There may be multiple data partition hierarchies that satisfy the above definition given a database decomposition P and a data partition graph $DPG(P,T_s)$. However, given any P and T_s, the existence of a DPH is guaranteed. This directly follows from the fact that the transitive reduction of any acyclic digraph G of the data partitions in P that is a total order of the data partitions satisfies the definition of $DPH(P,T_s)$.

In the remainder of the paper, the notation *DPH* refers to a particular data partition hierarchy chosen to base our hierarchical timestamp protocol. We say that a data partition D_i is *higher than* a data partition D_j, denoted as $D_i > D_j$, if there exists a directed path $D_j \rightarrow \cdots \rightarrow D_i$ in *DPH*. In general, we say that data partition D_i and data partition D_j are *related* if either $D_i = D_j$ or D_i and D_j are connected by a directed path.

2.1.3. Transaction Classification

Given a data partition hierarchy *DPH*, it is possible to classify transactions in T_s based on the data partition(s) they write into. Based on the definition of data partition hierarchy, if there exists a transaction $t \epsilon T_s$ which writes into more than one data partition, then all the data parti-

tions that it writes into must be connected by a directed path in the data partition hierarchy. Therefore, the rule for transaction classification is as follows:

Assign each update transaction t to the *highest* data partition D_i it writes into.

We denote t being assigned to D_i as $t \epsilon D_i$, and D_i is called the *home* data partition of t.

The following property establishes that the data partition that contains a data element that a transaction accesses must be related to the home data partition of that transaction in the data partition hierarchy. This property completes the background for establishing the hierarchical timestamp protocol to be described in the next section.

Property. Let $t \epsilon D_i$. Then the data partition that contains a data element that t accesses (read or write) is related to the D_i in DPH.

2.2. The Hierarchical Timestamp Protocol

The hierarchical timestamp protocol (HTS) is a concurrency control algorithm *parameterized* by a chosen data partition hierarchy *DPH* and the corresponding transaction classification. Like the conventional multi-version timestamp algorithm (MVTS), the HTS algorithm assigns a timestamp to a transaction t when it initiates. We will call this timestamp the initiation timestamp of t, denoted as $I(t)$. However, the timestamp the transaction actually uses to synchronize its accesses may be different from $I(t)$. In particular, the timestamp to access data elements in data partitions higher than t's home partition will normally be somewhat smaller than $I(t)$, while the reverse is true for accesses to lower partitions. The exact mechanism used to calculate these "access timestamps" will be such that the overall serializability is still preserved.

2.2.1. Basic Definitions

Two functions, called the *Decrement function*, or the function *DEC*, and the *Increment function*, or the function *INC*, are devised to compute the timestamps a transaction uses to access data partitions different from its own home partition. Function *DEC* is used for computing timestamps for accessing higher data partitions, and *INC* for lower partitions. These definitions

assume that a data partition hierarchy DPH is given.

Two values, $I(t)$ and $C(t)$, are associated with each transsaction t in the system. We assume that the timestamp assigned to a transaction is its *initiation time*, denoted as $I(t)$. Therefore every data access request issued by a transaction must occur after $I(t)$. Each transaction is also assigned a *commit timestamp* $C(t)>I(t)$. In typical cases, $C(t)$ is the time when t actually finishes; i.e., no data access request would normally be issued at time after $C(t)$ by transaction t. However there are exceptional occasions (as explained in *procedure C* later) when $C(t)$ is forced to be a smaller value than the time when t actually finishes. It is sufficient for our purpose to have $I(t)<C(t)$.

Definition. A transaction t is *active* at time m if $I(t)\leq m$ and $C(t)\geq m$.

Definition. The function I_i, defined for a data partition D_i, maps a time m to another time m', i.e., $m'=I_i(m)$, such that m' is the initiation time of the oldest active transaction assigned to D_i at time m. Formally,

$$I_i(m) = \begin{cases} m \text{ if } \textit{there exists no } t\epsilon D_i \textit{ active at time } m \\ Min(I(t)) \textit{ otherwise}, \text{ for } \textit{all } t\epsilon D_i \textit{ active at time } m. \end{cases}$$

Definition. Let the *Decrement function* $DEC_{i,j}$ be a function defined for a pair of data partitions D_i and D_j, where $D_j\geq D_i$. $DEC_{i,j}$ recursively maps a time to another time as follows.

$$DEC_{i,j}(m) = \begin{cases} m \text{ if } D_i=D_j \\ I_j(m) \text{ if } D_i\rightarrow D_j \textit{ is in DPH} \\ DEC_{k,j}(DEC_{i,k}(m)) \textit{ otherwise, where } D_i\rightarrow D_k\rightarrow\ldots\rightarrow D_j \textit{ is in DPH.} \end{cases}$$

That is, the function $DEC_{i,j}$ maps a time m in D_i to the initiation time, $DEC_{i,j}(m)$, of successively (i.e., along the critical path of DPH) the oldest active transaction assigned to D_j. For example, if the critical path between D_i and D_j is $D_i\rightarrow D_k\rightarrow D_j$, then $DEC_{i,j}(m)=I_j(I_k(m))$.

We will now describe the functions C_i and $INC_{j,i}$ that can be considered conceptually the *inverse* of functions I_i and $DEC_{i,j}$.

Definition. Let $C_i: m \rightarrow m'$ be a function which maps a time m to another m' where D_i is a data partition and $C_i(m)$ is determined as follows.

$$C_i(m) = \begin{cases} m \text{ if } \textit{there exists no } t \epsilon D_i \textit{ active at time } m, \\ Max(C(t)) \textit{ otherwise, for all } t \epsilon D_i \textit{ active at time } m. \end{cases}$$

That is, $C_i(m)$ is the *latest* commit time of all transactions assigned to D_i that started before or at time m.

While the *DEC* function maps a time in a lower partition to the initiation time of some transaction assigned to a higher partition, the *INC* function maps a time in a higher partition to the commit time of some transaction assigned to a lower partition:

Definition. The *Increment function*, defined for a pair of data partitions D_i and D_j, where $D_j \geq D_i$, denoted as $INC_{j,i}(m)$, is a function which maps a time value to another such that

$$INC_{j,i}(m) = \begin{cases} m \text{ if } D_i = D_j \\ C_j(m) \text{ if } D_i \rightarrow D_j \textit{ is in DPH} \\ INC_{k,i}(INC_{j,k}(m)) \textit{ otherwise, where } D_i \rightarrow \ldots . \rightarrow D_k \rightarrow D_j \textit{ is in DPH} \end{cases}$$

2.2.2. Hierarchical Timestamp Protocol

Now we introduce the hierarchical timestamp protocol, where synchronization timestamps are calculated according to functions *DEC* and *INC* above.

Hierarchical Timestamp Protocol For Update Transactions:

For every database access request from an update transaction $t \epsilon D_i$ for a data granule $d \epsilon D_j$, the following protocol is observed.

Protocol E (for accessing a partition *equal* to home partition)

If $D_j = D_i$, then (equivalent to the basic multi-version timestamping)

(1) If it is a read request, then grant the latest version before $I(t)$ of d, and leave $I(t)$ as the read timestamp of this version of d if its current read timestamp is smaller than $I(t)$.

(2) If it is a write request, then if the read timestamp of the latest version before $I(t)$ of d is smaller than $I(t)$, then create a new version of d with version number $I(t)$. Otherwise abort t.

Protocol H (For accessing Higher partitions)

If $D_j > D_i$ then grant t access to the latest version before $DEC_{i,j}(I(t))$ of d.

Protocol L (For accessing Lower partitions)

If $D_i > D_j$, then

(1) If it is a read request, then grant the latest version before $INC_{i,j}(I(t))$ of d, and leave $INC_{i,j}(I(t))$ as the read timestamp of this version of d if its current read timestamp is smaller than $INC_{i,j}(I(t))$.

(2) If it is a write request, then if the read timestamp of the latest version before $INC_{i,j}(I(t))$ of d is smaller than $INC_{i,j}(I(t))$, then create a new version of d with version number $INC_{i,j}(I(t))$. Otherwise abort t.

2.2.3. Implementing HTS Protocols

To implement the HTS protocol, we must specify how $C(t)$ is assigned and how $I_i(m)$ and $C_i(m)$ are computed operationally. We will first introduce two procedures: (1) $ProcC_i$, used for computing $C_i(m)$, and (2) $ProcI_i$, used for computing $I_i(m)$. The procedure $ProcC_i(m)$ adds a constant time quantum $q_i > 0$, generic to a data partition D_i, to the argument value. In addition, it remembers that the procedure has been invoked for time value m so as to constrain future assignment of $C(t)$ for $t \epsilon D_i$. The procedure $ProcI_i(m)$ computes the initiation timestamp of the oldest active transaction in D_i at time m. With $ProcC$ and $ProcI$ defined, the timestamps used in the HTS protocols are computed as follows. Let the path in DPH between two data partitions D_i and D_j be $D_i \rightarrow D_k \rightarrow \cdots \rightarrow D_j$. The timestamp $DEC_{i,j}(m)$ used in Protocol H is computed as $ProcI_j(...(ProcI_k(m)...)$, and that used in Protocol L is computed as $ProcC_k(...(ProcC_j(m)...)$. These procedures are consistent with the definitions of the functions DEC and INC given in the

previous subsection.

These procedures are described as follows:

$ProcC_i(m)$: Procedure.

If $m+q_i$ is less than current time, then return ("Abort Requestor");

Mark m; /* remembers that $ProcC_i$ has been invoked with argument m */

Create a pseudo transaction $t' \epsilon D_i$ s.t. $I(t')=m$ and $C(t')=m+q_i$.

Return $(m+q_i)$.

The commit timestamp $C(t)$ is computed for $t \epsilon D_i$ where t *finishes* at time m as follows:

If there exists an invocation of $ProcC_i(m')$ s.t. $I(t)<m'<m-q_i$ then $C(t)$ is asssigned the minimum of $m'+q_i$ over all such m'. Otherwise $C(t)=m$.

In other words, if $ProcC_i$ has not been invoked in D_i during the life time of t then the commit timestamp of $t \epsilon D_i$ is simply t's finish time. When such invocations exist, and at least one of them was invoked with an argument value which is at least q_i time units before t's finishing time (which also implies that t is at least q_i long), then t's commit timestamp is pushed backwards. It is noted that If the time quantum q_i's are selected to be large enough such that most of the transactions in D_i that require accesses to lower data partitions would do so within q_i units of time after they start, the chance of having to abort such transactions due to aborted $ProcC_i$ procedures would be relatively small, and virtually all $C(t)$ would be the same as the actual finish time of t.

The procedure $ProcI_i(m)$ is defined as follows:

$ProcI_i(m)$:

If there exists unfinished transaction t at time m s.t. its finish time is unknown (i.e., t is not finished at the time $ProcI_i(m)$ is invoked,) and it is known that $C(t)$ is samller than m, then suspend requestor till t finishes;

Compute the initiation time of the oldest active transaction at time m as m';

Return (m');

It is noted that $ProcI_i$ would suspend requestor only if there exists a previous invocation of $C_i(m')$ in D_i for some $m' < m - q_i$ s.t. transactions started before m' have not finished by the time $ProcI_i(m)$ is invoked. Note that $ProcI_i(m)$ is always invoked at a time later than m. Therefore, for appropriately large q_i's the chance of an invocation of $ProcI_i(m)$ needing to block is very small.

To ensure that the procedure $I_i(m)$ is implementable, we argue that when $I_i(m)$ is invoked to compute an access timestamp, all information needed to correctly compute $I_i(m)$ is available. Since $I_i(m)$ is always invoked at a time later than m, it is sufficient to argue that at time m, all information needed to correctly compute $I_i(m)$ is available:

At time m, all regular transactions in D_i with $I(t) < m$ would have been known. Therefore we only need to consider pseudo transactions. Pseudo transactions t' with $I(t') < m$ would be inserted at time $mf > m$ only if $I(t') > mf - q_i > m - q_i$. If such t' is inserted as a result of computing $INC_{i,j}(x)$ then there must exist a transaction t'' in D_i s.t. $I(t'') = I(t')$ and at time mf it is still unfinished as it is making a request to lower data segment at time mf. Therefore the effect of the pseudo t' on $I_i(m)$ would be dominated by that of t'' which is known by time m. If such t' is inserted as a result of computing $INC_{k,j}(x)$ where D_i is on the path between D_k and D_j, then due to the behavior of $ProcC_k$ the argument that $ProcC_i$ receives must be greater than the time when $ProcC_i$ is invoked. Therefore a pseudo transaction t' with $I(t') < m$ must be known by time m. Therefore all regular and pseudo transactions that can influence the value of $I_i(m)$ are known by m. For all transactions known to be unfinished at time m whose commit timestamps would be smaller than m are also known by time m due to the rule for assigning $C(t)$ to values other than t's actual finish time. Therefore all information on transactions needed to compute $I_i(m)$ is known by time m.

2.2.4. Remarks

To summarize, several observations are made of this set of protocols:

(1) If the database partition hierarchy DPH consists of a single data partition, then only Protocol E will apply, and the hierarchical timestamp algorithm is reduced to the conventional MVTS algorithm.

(2) Since no transaction will write a data partition higher than its own home partition, Protocol H needs to cover only read accesses. More importantly, Protocol H is "cheaper" than either Protocol E or Protocol L, since it does not require timestamping the data element accessed. The key benefit of the HTS algorithm comes from choosing a database partition such that Protocol H is used much more than Protocol L.

(3) Protocol L uses timestamps which are different from the timestamps of the accessing transactions. Since $INC_{i,j}(I(t)) \geq I(t)$, Protocol L is the most "expensive" among the three, as it tends to increase the chance for transactions in lower data partitions to abort. In fact, the larger is q_i, the better off are transactions in D_i when they access lower data partitions, as there is less probability for them to abort; and the worse off are the transactions belonging to lower data partitions. Offering tradeoffs is intrinsic to hierarchical timestamping. System designers may reflect their perceived priorities among all transactions by their choice of such parameters as q_i.

3. Proof of Correctness

3.1. Proof Structure

To show that the above algorithm is correct, one must show that serializability is enforced. We first formulate the problem of testing serializability, and then apply it to the HTS algorithm.

Definition. A *schedule* is a sequence of steps, each of which is in the form of a tuple <transaction id, action, version of a data granule>. The action can be read (r) or write (w). The version of a data granule is denoted as d^v, where d indicates the data granule and v indicates the version. If the action is write, then the version of the data granule included in the step is created by the transaction. If the action is read, then the transaction reads the version of the data

granule indicated in the tuple. An example of a schedule is $<t_1,w,d^i>$, $<t_2,r,d^i>$, $<t_2,w,d^j>$, $<t_3,r,d^j>$.

Definition. Assume that a version order, denoted as $<<$, is given. A version j of a data element d is the *predecessor* of a version k of d if $j << k$ and there exists no version i of d such that $j << i << k$.

Definition. A *transaction dependency graph* of a schedule S is a directed graph, denoted as TG(S), where the nodes are the transactions in S and the arcs, representing *direct dependencies* between transactions, exist according to the following rule:

$t_2 \rightarrow t_1$ is in TG(S) iff

(1) $<t_1,w,d^j>$ and $<t_2,r,d^j>$ are in S for some d^j, or

(2) $<t_1,r,d^j>$ and $<t_2,w,d^k>$ are in S for some d^j, d^k where d^j is the predecessor of d^k. (d^j is a *predecessor* of d^k if $j < k$ and there exists no d^o such that $j < o < k$.)

The following theorem is adapted from the 1-Serializability Theorem proven in [Bernstein83]:

Theorem. Given a version order $<<$, a schedule S is serializable if TG(S) is acyclic.

Given the above rule for testing for serializability, the following steps are devised to prove correctness of the HTS algorithm:

(1) Define a relation *topologically follows*, denoted as $=>$, between any pair of transactions that are assigned to related data partitions in DPH.

(2) Show that direct dependencies may occur only between transactions that are assigned to partitions that are related in DPH.

(3) Show that if a schedule S enforces the relation $=>$ on all direct transaction dependencies (i.e., $t_2 \rightarrow t_1 \epsilon TG(S)$ only if $t_2 => t_1$.) then the transaction dependency graph TG(S) has no cycle.

(4) Show that the HTS algorithm produces only schedules that enforce the relation $=>$ on all direct transaction dependencies.

The proof structure is important in understanding how the implementation details of the HTS protocols may be modified without having to reconstruct proofs from scratch. In particular, much of the proof is accomplished by referencing solely the *definitions* of the functions *DEC* and *INC*, not their implementations. The only exception is in the last step (i.e., step (4)), a key step of the proof in this paper. Only a small portion of its proof directly cites the implementation of the protocol. This portion will be clearly identified.

3.2. The topologically-follows Relation and its Properties

We now define *topologically follows*, which assumes a given data partition hierarchy.

Definition. A relation *topologically follows*, denoted as $=>$, is defined for a pair of transactions t_1, t_2 where $t_1 \epsilon D_i$, $t_2 \epsilon D_j$, D_i and D_j are related in the chosen data partition hierarchy. We say that t_1 *topologically follows* t_2 (or $t_1 => t_2$) iff

(1) if $D_i = D_j$ then $I(t_1) > I(t_2)$,

(2) if $D_i > D_j$ then $I(t_1) \geq DEC_{j,i}(I(t_2))$,

(3) if $D_i < D_j$ then $I(t_2) < DEC_{i,j}(I(t_1))$.

Clearly, $=>$ is defined only between transactions that belong to data partitions that are related in DPH, because otherwise the *DEC* function is undefined.

Lemma. Given a DPH and let $t_1 \epsilon D_i$ and $t_2 \epsilon D_j$. If $t_2 \rightarrow t_1$, then D_i and D_j are related in DPH.

Proof. Let $d \epsilon D_k$ be the contended data element that causes $t_2 \rightarrow t_1$. Since at least one transaction writes d, without loss of generality, let's assume it to be t_1 that writes d. Then by the definition of data partition graph based on transaction analysis, either $D_k \rightarrow D_i \epsilon DSG$ or $D_k = D_i$. Since t_2 must at least read d, therefore either $D_j \rightarrow D_k \epsilon DSG$, or $D_j = D_k$. This means that $D_j \rightarrow (=) D_k \rightarrow (=) D_i \epsilon DSG$. This means that if $D_j \neq D_i$, then there exists a directed path between

D_j and D_i in DSG. By the derivation of the data partition hierarchy, there must be a directed path between D_j and D_i in DPH, which means that D_j and D_i are related in DPH. *Q.E.D.*

In [Hsu86] the following theorem concerning the partial ordering effect of the relation $=>$ was proven:

Theorem 1. Let TG(S) be a transaction dependency graph of a set of update transactions run on a database. Let S enforce the synchronization rule that, given a DPH, $t_2 \rightarrow t_1 \epsilon TG(S)$ only if $t_2 => t_1$. Then TG(S) has no cycles.

In fact, a statement stronger than the above was proven in [Hsu86]. What was proven was that if S enforces the rule that $t_2 \rightarrow t_1 \epsilon TG(S)$ only if $t_2 =>_{TS,AL} t_1$ then TG(S) has no cycle, where $=>_{TS,AL}$ is defined to be *topologically follows* with respect to two functions TS and AL_{ij}, with the requirements that TS maps each transaction to a unique time value, and AL_{ij} satisfies the following properties:

(1) Composability: $AL_{kj}(AL_{ik}(m))=AL_{ij}(m)$ for all times m and for all D_i, D_k and D_j where $D_j > D_k > D_i$.

(2) Non-decreasing: $AL_{ij}(m) \geq AL_{ij}(m')$ for all D_i and D_j where $D_j > D_i$ and for all times m, m' where $m > m'$.

In essence, the relation $=>$ defined here is an instance of $=>_{TS,AL}$ where the function TS has been assigned the initiation timestamping I of transactions, and AL_{ij} has been assigned $DEC_{i,j}$. It is easy to verify that I and $DEC_{i,j}$ satisfy the requirements for TS and AL.

3.3. HTS Enforces topologically-follows

Given the above theorems stating that the relation $=>$ can be used as a vehicle for ordering transactions for concurrency control purposes, the following theorem completes our proof of correctness.

Theorem 2. Let S be a schedule that is permitted by the hierarchical timestamp protocol. Then $t_2 \rightarrow t_1 \epsilon TG(S)$ only if $t_2 => t_1$.

Proof. Let $t_2 \epsilon D_j$ and $t_1 \epsilon D_i$. $t_2 \rightarrow t_1$ is translated into the following three cases:

(1) t_2 reads a data element $d \epsilon D_k$ written (created) by t_1.

(2) t_2 writes (i.e., creates a new version of) a data element $d \epsilon D_k$ whose predecessor was read by t_1.

(3) t_2 writes (creates a new version of) a data element $d \epsilon D_k$ whose predecessor was written (created) by t_1 and the version created by t_2 is read by another transaction.

What has to be shown is that in all three cases $t_2 => t_1$ holds. In all cases, the following two lemmas which bind the functions *DEC* and *INC* together are used to transform the timing relationship imposed by the protocol to that defined by the relation $=>$:

Lemma 1. $DEC_{i,j}(INC_{j,i}(m)) \leq m$, for all $D_j > D_i$ in *DPH*.

Lemma 2. $DEC_{j,i}(INC_{i,j}(m)+\epsilon) > m$, for all $D_j > D_i$ in *DPH*, for all $\epsilon > 0$.

We will first prove these two lemmas.

Lemma 1.

Proof. We first prove that $I_i(C_i(m)) \leq m$ for all i,m. If there exists no transaction active at time m in D_i then $C_i(m)=m$, and $I_i(C_i(m))=I_i(m)=m$. If there exists at least one transaction active at time m in D_i, let the transaction with the largest commit timestamp among all that are active at time m be t_o, then $C_i(m)=C(t_o)$. Therefore, at time $C_i(m)$, there is at least one transaction active (since at least t_o is active), and the oldest active transaction at that time must be at least as old as t_o, where t_o was active at time m, which means $I(t_o) \leq m$. Therefore $I_i(C_i(m)) \leq I(t_o) \leq m$. Therefore we conclude that $I_i(C_i(m)) \leq m$ for all i,m. Let the critical path between D_i and D_j be $D_i \rightarrow D_{i1} \rightarrow D_{i2} \rightarrow \cdots \rightarrow D_{in} \rightarrow D_j$. Then by expanding according to the definition of *DEC* and *INC*, $DEC_{i,j}(INC_{j,i}(m)) = I_j(I_{in}(....(I_{i2}(I_{i1}(C_{i2}(...(C_{in}(C_j(m)))...)))))..))$. Let $C_{i2}(...(C_{in}(C_j(m)))...)=m_{i2}$, then we have $I_{i1}(C_{i1}(m_{i2})) \leq m_{i2}$. Therefore $DEC_{i,j}(INC_{j,i}(m)) \leq I_j(I_{in}(...(I_{i2}(m_{i2})) = I_j(I_{in}(...(I_{i2}(C_{i2}(...(C_{in}(C_j(m)))...)$. Continuing with the same reasoning, we have $DEC_{i,j}(INC_{j,i}(m)) \leq I_j(C_j(m)) \leq m$. *Q.E.D.*

Lemma 2.

Proof. We first prove that $I_i(C_i(m))+\epsilon)>m$ for all i,m. There are two cases.

(a) If no transaction is active at time m in D_i then $C_i(m)=m$. Therefore at time $C_i(m)+\epsilon$ the oldest active transaction, if any, cannot have started on or before m. Therefore $I_i(C_i(m)+\epsilon)>m$.

(b) If some transaction is active at time m in D_i, then let t_o be defined the same way as was in the proof of *Lemma 1*, we have $C_i(m)=C(t_o)$. Therefore at time $C(t_o)+\epsilon$, no transaction active at time m is still active; That is, $I_i(C(t_o)+\epsilon)>m$, or $I_i(C_i(m)+\epsilon)>m$. Combining (a) and (b) one concludes that $I_i(C_i(m))>m$ for all i,m. Let the path between D_i and D_j be $D_i \rightarrow D_{i1} \rightarrow D_{i2} \rightarrow D_{i3} \rightarrow \cdots \rightarrow D_{in} \rightarrow D_j$. Let $C_{i3}(...(C_{in}(C_j(m)))...)=m_{i3}$ and let $C_{i2}(Ci3...(C_{in}(C_j(m)))...)=m_{i2}$. That is, $m_{i2}=C_{i2}(m_{i3})$. By the above conclusion, we have $I_{i1}(C_{i1}(C_{i2}(m_{i3}))+\epsilon)>C_{i2}(m_{i3})$. That is, $I_{i1}(C_{i1}(C_{i2}(m_{i3}))+\epsilon)\geq C_{i2}(m_{i3})+\epsilon'$. Therefore, applying I_{i2} to both sides, $I_{i2}(I_{i1}(C_{i1}(C_{i2}(m_{i3}))+\epsilon))\geq I_{i2}(C_{i2}(m_{i3})+\epsilon') > m_{i3}$. Continuing with the same reasoning one derives $DEC_{i,j}(INC_{j,i}(m)+\epsilon)= I_j(I_{in}(....(I_{i1}(C_{i1}(...(C_{in}(C_j(m))...)+\epsilon)..)) > m$. *Q.E.D.*.

Now we are ready to complete the proof for *Theorem 2*. For the three cases identified previously:

(1) In this case, $D_i \geq D_k$. Since D_j and D_i must be related, we have three cases and for each case observing the hierarchical timestamp protocol leads to the conclusion that $t_2 => t_1$. (a) $D_j \geq D_i \geq D_k$. In this case, obeying HTS leads to the assertion: $INC_{j,k}(I(t_2))>INC_{i,k}(I(t_1))$ **(0.1)**. Therefore $INC_{j,k}(I(t_2))\geq INC_{i,k}(I(t_1))+\epsilon$. Applying function $DEC_{k,i}$ to both sides we obtain $DEC_{k,i}(INC_{j,k}(I(t_2)))\geq DEC_{k,i}(INC_{i,k}(I(t_1))+\epsilon)$ **(0.2)**. From Lemma 2 above the right hand side of **(0.2)** is greater than $I(t_1)$. Applying function $DEC_{i,j}$ to both sides we have $DEC_{i,j}(DEC_{k,i}(INC_{j,k}(I(t_2))))\geq DEC_{i,j}(DEC_{k,i}(INC_{i,k}(I(t_1))+\epsilon) \geq DEC_{i,j}(I(t1))$. However, the left-hand side of the above expression = $DEC_{k,j}(INC_{j,k}(I(t_2)))$, which by Lemma 1 is less than or equal to $I(t_2)$. Therefore $I(t_2)\geq DEC_{i,j}(I(t_1))$, which means $t_2 => t_1$. (b) $D_i > D_j \geq D_k$. In this case **(0.1)** holds also. Apply $DEC_{k,i}$ to both side of **(0.1)** we have $DEC_{j,i}(Aj(INC_{j,k}(I(t_2))))\geq DEC_{k,i}(INC_{i,k}(I(t1))+\epsilon)$. Right hand side is $>I(t_1)$. Left hand

side is $\leq DEC_{j,i}(I(t_2))$. Therefore $DEC_{j,i}I(t_2))>I(t_1)$, which means $t_2=>t_1$. (c) $D_i \geq D_k > D_j$. In this case $INC_{i,k}(I(t_1))<DEC_{j,k}(I(t_2))$. i.e., $INC_{i,k}(I(t_1))+\epsilon \leq DEC_{j,k}(I(t_2))$. Apply $DEC_{k,i}$ to both sides we have $I(t_1)<DEC_{j,i}(I(t_2))$. i.e., $t_2=>t_1$.

(2) In this case, $D_j \geq D_k$. By the same reasoning in (1) we can show $t_2=>t_1$ for the cases of $D_j \geq D_i \geq D_k$ and $D_i > D_j \geq D_k$. Now we consider the cases where $D_j \geq D_k > D_i$. Let the time at which t_2 performs writing d be m. Consider two cases. (a) If t_1's request to read d arrives at or before time m, which means t_1 had started at or before m, i.e., $I(t_1) \leq m$ and $DEC_{i,k}(I(t_1)) \leq I_k(m) \leq m$ **(0.3)**. (a.1) $D_j > D_k$. In this case, by the Protocol L $INC_{j,k}(I(t_2)) \geq m$, for otherwise the write request would have been rejected (in $ProcC_j(I(t_2))$). Combine this with **(0.3)** we have $INC_{j,k}(I(t_2)) \geq DEC_{i,k}(I(t_1))$. Apply $DEC_{k,j}$ to both sides we get $I(t_2) \geq DEC_{i,j}(I(t_1))$, i.e., $t_2=>t_1$. (a.2) $D_j=D_k$. In this case, $INC_{j,k}(I(t_2))=I(t_2)$. Since t_2 makes a request at time m, t_2 is active at time m (and therefore $I_k(m) \leq I(t_2)$), except when t_2 satisfies conditions under which its commit timestamp is to be pushed backwards to be before m. However when such conditions hold the $ProcI_k(m')$ procedure for computing $DEC_{i,k}(I(t_1))$ for $m' < m$ would have to block till t_2 finishes, which is after m, contradictory to the given that t_1's request to read d arrives before m. Therefore $I_k(m) \leq I(t_2)$. Therefore $INC_{j,k}(I(t_2)) \geq I_k(m) \geq DEC_{i,k}(I(t_1))$. By same reasoning in (a.1) we have $t_2=>t_1$. (b) If t_1's request to read d arrives after m, then the version created by t_2 already existed, and by Protocol H if t_1 chooses to read a version before that created by t_2 it must be $INC_{j,k}(I(t_2)) \geq DEC_{i,k}(I(t_1))$. Therefore $t_2=>t_1$. (b)

(3) In this case, it suffices to prove that $t_2=>t_1$ when t_2 creates a new version of d whose predecessor is created by t_1. Since in this case $D_i \geq D_k$ and $D_j \geq D_k$, the two relevant cases are already shown in (1) and therefore $t_2=>t_1$. *Q.E.D.*

Proof of Theorem 2 is largely independent of the implementation procedures for *ProcC* and *ProcI* except for arguments in (2)(a). The dependency arises due to our desire to allow the H protocol in HTS *not to have to leave read timestamps*. If HTS is modified such that Protocol H

also required read timestamps, then the proof can be constructed entirely from the definitions of functions *DEC* and *INC* and their properties without resorting to implementations.

4. Discussion of Performance

Performance of a concurrency control algorithm can be assessed along two dimensions. One is the computational overhead, such as setting locks and timestamping data elements. The other is the level of concurrency, or "optimality" of the algorithm. Along both dimensions, the HTS algorithm potentially performs better than the MVTS algorithm, if a data partition hierarchy can be found such that the frequency of accesses from transactions assigned to lower partitions to data elements contained in higher data partitions dominates that of the reverse direction. In other words, if the data partition hierarchy is chosen such that Protocol H (the "cheap" protocol) is used much more often than Protocol L (the "expensive" protocol,) then a net saving may be achieved to warrant the use of this algorithm.

The HTS algorithm requires a transaction analysis and the overhead of maintaining selective information on transaction initiation and commit times to enable computation of the A function and enforcement of the B function. However, the fact that Protocol H does not require timestamping the data elements makes it very attractive for situations where transactions assigned to a lower partition need to access a certain quantity of data in a higher data partitions. Therefore the HTS algorithm can potentially produce a net saving in computational overhead if Protocol H is used "often enough."

Now we analyze the optimality aspect of the algorithm. In [Kung79], optimality of a concurrency control algorithm is defined as the degree to which the algorithm allows for "serializable input schedules" to proceed without being interrupted. Intuitively, an input schedule, which is an interleaved sequence of read and write requests issued by a set of transactions, is multi-version-serializable (MVSR) if there exists a way of assigning versions to these read and write requests such that the resulting transaction dependency graph is acyclic. For reasons of implementability,

no known multi-version concurrency control algorithm will allow all MVSR input schedules to proceed without interruption. However, the degree to which an algorithm allows MVSR schedules to proceed without interruption can be used as a useful measure of level of concurrency.

In comparing the HTS algorithm to the MVTS algorithm, it can be argued that if Protocol H is used much more frequently than Protocol L, then the expected number of restarts (i.e., transaction aborts) under the HTS algorithm would be smaller than that under the MVTS algorithm. While further studies are needed to quantify this statement, the following theorem confirms that, in the extreme case when Protocol L is not used at all, the HTS algorithm dominates the MVTS algorithm in terms of optimality.

Lemma 3. If the data partition graph $DPG(P,T_u)$ of some database decomposition P is acyclic, then among the set of correct data partition hierarchies $DPH(P,T_u)$, there exists DPH such that Protocol L is not needed to process update transactions in T_u. (This can be shown by assigning to DPH the partial order of data partitions exhibited in $DPG(P,T_u)$, and since no path exists in $DPG(P,T_u)$ from a higher data partition to a lower data partition, there exists no accesses from transactions assigned to higher data partitions to the lower data partitions, and therefore Protocol L is never needed.)

Theorem 3. Let P be a database decomposition and $DPG(P,T_u)$ be acyclic. Let the data partition hierarchy DPH be such that Protocol L is never used in processing transactions in T_u. Let $S(MVTS)$ denote the set of serializable input schedules that are allowed for execution by MVTS without interruption, and $S(HTS)$ that by the HTS algorithm under the data partition hierarchy DPH where *both* Protocol H and Protocol E are used. Then $S(MVTS)$ is a subset of $S(HTS)$.

Proof. We must show that (1) any schedule allowed by MVTS must be allowed by HTS, and (2) there exists at least one schedule allowed by HTS that is not allowed by MVTS.

(1) Let a schedule $S \epsilon S(MVTS)$. Want to show that $S \epsilon S(HTS)$. Suppose this is not true. Then there must exist a step (t_i,w,d) in S that HTS will abort. We will show that this is

impossible. Let $t_i \epsilon D_1$. Let $(t_j,r,d)<(t_i,w,d)$ denote that fact that step (t_j,r,d) is before (t_i,w,d) in S. Since Protocol L is never used, the timestamp used to synchronize (t_i,w,d) by HTS must be $I(t_i)$. Therefore it suffices to show that there exists no $(t_j,r,d)<(t_i,w,d)$ in S that will cause the HTS algorithm to leave a read timestamp greater than $I(ti)$ with d. This is true because for all (t_j,r,d) where $t_j \epsilon D_2$, either $D_1=D_2$ or $D_1>D_2$. In the latter case, no read timestamp is left by HTS. In the former case, the read timestamp left by HTS is equal to that left by MVTS, which must be smaller than $I(t_i)$.

(2) Consider the following schedule: $S=(t_1,r,a),(t_2,r,a),(t_2,w,a),(t_2w,,b)$, where $t_1 \epsilon D_i$, $a \epsilon D_i$, $t_2 \epsilon D_j$, $b \epsilon D_j$, $D_i>D_j$ in DPH, and $I(t_1)<I(t_2)$. Clearly $S \epsilon S(HTS)$ and S is not a member of $S(MVTS)$. *Q.E.D.*

The above result implies that, given a database application, if the set of update transactions T_s can be designed in such a way that a non-trivial database decomposition P can be found such that the data partition graph $DPG(P,T_s)$ is acyclic, then the HTS algorithm will definitely perform better than the MVTS algorithm in terms of level of concurrency. When such design cannot be achieved, but a design is found that enables construction of a data partition hierarchy where usage of Protocol H is much higher than that of Protocol L, the HTS algorithm can still be expected to perform better than the MVTS algorithm.

In [Hsu83b], a simple analytical model was established for an application of the HTS algorithm to a simple two-data-partition database. In the analysis, it is assumed that the higher data partition is more heavily contended for by transactions than the lower data partition. The simplicity of the case enables the use of simple analysis to obtain formuli for the rate of blocking under two-phase locking (2PL) due to conflict in the higher data partition. algorithm and the rate of abort under the HTS algorithm. The analysis makes assumptions that are in general in favor of the 2PL approach and discriminates against the HTS approach. The purpose is to demonstrate the effect of the HTS algorithm in relieving contention in the higher data partition, presumably the much more heavily contended data partition in our case The result, reported in [Hsu83b],

shows that, in general, the rate of blocking under 2PL is proportional to $a_1 * MP_1^2 + a_2 * MP_1 * MP_2$, where MP_1 is the number of type-1 transactions in the system at any time and type-1 transactions are those assigned to the higher data partition, MP_2 that of type-2 transactions and type-2 transactions are those assigned to the lower data partition, and ai is the number of data elements in D_1 accessed by a typical transaction assigned to D_i, where i=1,2. In contrast, the rate of abort under HTS is proportional only to $a_1 * MP_1^2$. While the absolute values depend on other parameters, one can conclude that HTS would be preferred when $a_2 * MP_2$ is large.

5. Conclusion

The Hierarchical Timestamp Algorithm can be considered as a generalized timestamp algorithm parameterized by a data partition hierarchy constructed via transaction analysis. It relies on an understanding of the structural disciplines of an application system and represents an attempt to take advantage of these disciplines. By being sensitive to the existence of such structures and being flexible in manipulating the basic control tools (e.g., timestamps, locks) this structural approach to concurrency control holds promise for improving applications or activities in databases where the level of concurrency is vital to system performance.

The thrust of this paper thus goes beyond proposing a new concurrency control algorithm. It demonstrates the potential benefit of exploiting the knowledge and the structure of the application systems to implement more efficient and more tailored concurrency control mechanism. It also points out an area of research which examines the problem of *how to design transactions* so that concurrency control can be more efficient without compromising the key requirements on the data. It is believed that transaction design performed with concurrency control problems in mind could *produce* a structure of applications that is significantly less costly for concurrency control to be implemented.

6. References

[Bayer80]
Bayer, R., Heller, H., and Reiser, A. Parallelism and recovery in database systems. ACM Trans. Database Syst., 5, 2, June 1980.

[Bernstein80]
Bernstein, P.A., Shipman, D.W., and Rothnie, J.B. Concurrency control in a system for distributed databases (SDD-1). ACM Trans. Database Syst., 5, 1, March 1980.

[Bernstein81]
Bernstein. P.A and Goodman, N. Concurrency control in distributed database systems. ACM Comp. Surv., 13, 2, June 1981.

[Bernstein83]
Bernstein, P.A. and Goodman N. Multi-version concurrency control - theory and algorithms. ACM Trans. Database Syst., 8, 4, December 1983.

[Carey86]
Carey, M. and Muhanna, W.A. The performance of multi-version concurrency control algorithms. To appear in *ACM Transactions on Computer Systems.*

[Chan82]
Chan, A. et. al. The implementation of an integrated concurrency control and recovery scheme. ACM SIGMOD Conference Proceedings, 1982.

[Chan85]
Chan, A., Gray, R. Implementing distributed read-only transactions. IEEE Trans. Softw. Engr., SE-11, 2, February 1985.

[Eswaran76]
Eswaran, K.P., Gray, J.N., Lorie, R.A. and Traiger, I.L. The notions of consistency and predicate locks in a database systems. Comm. ACM, 19, 11, November 1976.

[Garcia-Molina82]
Garcia-Molina, H. and Wiederhold, G. Read-only transactions in a distributed database. ACM Trans Database Syst., 7, 2, June 1982

[Gray78]
Gray, J.N. Notes on database operating systems. In *Operating Systems - An Advanced Course.* R. Bayer, R.M. Graham, B.G. Seegmuller, Eds., Springer-Verlag, 1978.

[Harder86]
Harder, T. and Petry, E. Evaluation of a multiple version scheme for concurrency control. Technical Report, University Kaiserslautern, West Germany, January 1986.

[Hsu83]
Hsu, M and Madnick, S.E. Hierarchical database decomposition: a technique for database concurrency control. Proceedings of 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 1983.

[Hsu83b]
Hsu, M. The Hierarchical Database Decomposition approach to Concurrency Control. Ph.D. Thesis, Sloan School, M.I T., Cambridge, MA, 1983.

[Hsu86]
Hsu, M. and Chan, A Partitioned two-phase locking. To appear in *ACM Transactions on Database Systems*

[Kedem83]
kedem, Z. and Silberschatz, A Locking protocols: from exclusive to shared locks. Journal of ACM, 30, 4, October 1983

[Kung79]
Kung, H.T. and Papadimitriou, C.H. An optimality theory of concurrency control for databases. ACM SIGMOD Conference Proceedings, 1979.

[Papadimitriou84]
Papadimitriou, C.H. and Kanellakis, P.C. On concurrency control by multiple versions. ACM Trans. Database Syst., 9, 1, March 1984.

[Reed78]
Reed, D.P. Naming and synchronization in a decentralized computer system. Ph.D. dissertation, Dept. of Electrical Engineering and Computer Science, M.I.T., Cambridge, Mass. September 1978.

[Reed79]
Reed, D.P., Implementing atomic actions. In proceedings 7th ACM Symposium on Operating Systems Principles, December 1979.

[Silberschatz80]
Silberschatz, A. and Kedem, Z. Consistency in hierarchical database systems. Journal of ACM, 27, 1, January 1980.

[Silberschatz82]
Silberschatz, A., Kedem, Z. A family of locking protocols for database systems that are modeled by directed graphs. IEEE Trans. Softw. Engr., 8,6, November 1982.

[Stearns81]
Stearns, R. and Rosenkrantz, D. Distributed database concurrency control using before-values. ACM SIGMOD Conference Proceedings, 1981.

[Viemont82]
Viemont, Y.H. and Gardarin, G.J. A distributed concurrency control algorithm based on transaction commit ordering. Proceedings of Fault Tolerance Computer Systems, June 1982.